RECONFIGURABLE SIMD MULTIPLY-AND-ACCUMULATOR AND EFFICIENT ON-CHIP TRAINING OF DEEP NEURAL NETWORKS

Daewoo Kim

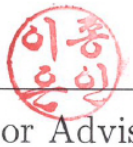
Department of Computer Science and Engineering
Graduate school of UNIST

Reconfigurable SIMD Multiply-and-Accumulator and Efficient On-Chip Training of Deep Neural Networks

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Daewoo Kim

06.14.2018
Approved by



Major Advisor
Jongeun Lee

Reconfigurable SIMD Multiply-and-Accumulator and Efficient On-Chip Training of Deep Neural Networks

Daewoo Kim

This certifies that the thesis of Daewoo Kim is approved.

06.14.2018

Thesis Supervisor: Jongeun Lee

Woongki Baek: Thesis Committee Member #1

Seokhyeong Kang: Thesis Committee Member #2

Abstract

In FPGA (Field-Programmable Gate Arrays) field, most of researchers mainly focus on inference acceleration of deep neural network although learning acceleration of deep neural network is also important in terms of machine learning. Thus, this paper presents learning-inference architecture to support both learning and inference acceleration of deep neural network. While inferencing, Double MAC is performed to double performance by packing two multiply-accumulate (MAC) operations into one DSP block like SIMD (Single Instruction Multiple Data) operation. On the other hand, in case of learning, we use one DSP block for one MAC operation to more accurate computation. We implement a learning-inference accelerator using Double MAC on a ZC706 FPGA board with Caffe framework.

Contents

Contents v

List of Figures vi

List of Tables vi

I. INTRODUCTION 1

II. BACKGROUND AND PREVIOUS WORK 3

2.1 Background . . . 3

2.1.1 Fixed Point Arithmetic . . . 3

2.1.2 DSP Block in State of The Art FPGAs . . . 3

2.2 Previous Work . . . 4

2.2.1 The Datapath of Our Architecture . . . 4

2.2.2 The Method of Modification . . . 4

III. MAPPING DEEP NEURAL NETWORK TO FPGA 6

3.1 Our Architecture . . . 6

3.1.1 Implementation of MAC Array Using Double MAC . . . 6

3.1.2 The Datapath of Double MAC for Supporting Both Training and Inference 6

3.1.3 RTL-HLS Hybrid Architecture . . . 8

3.1.4 System Block Diagram . . . 10

3.2 Caffe Framework on FPGA . . . 10

3.2.1 Caffe Structure with FPGA . . . 10

3.2.2 Converting GEMM to Common Type of Matrix Multiplication . . . 11

3.2.3 Weights Update . . . 12

3.2.4 Tiled Matrix Multiplication . . . 12

IV. EXPERIMENTS 15

4.1 Experiment Setup . . . 15

4.2 Synthesis Result 15
4.3 Validation of Performance Model 16
4.4 Performance 17

V. FUTURE WORK **19**

VI. CONCLUSION **20**

References **21**

List of Figures

2.1 Our proposed Double MAC architecure 4

3.1 Our proposed Double MAC array for supporting 8-bit and 16-bit 7
3.2 Our buffer structure for Double MAC array 9
3.3 System block diagram 10
3.4 Caffe structure with FPGA 11
3.5 Two-level tiled matrix multiplication scheme with parameters (T and μ) 13
3.6 Matrix multiplication code in C with tile parameters (T and μ). 13

4.1 Xilinx Zynq-7000 SoC ZC706 Evaluation Kit 16
4.2 Caffe applications profiling 17

List of Tables

3.1 Vector concatenation 8
3.2 Matrix multiplication parameters 12

4.1 Resource utilization 16

CHAPTER I

INTRODUCTION

Arguably one of the most unique features of Deep Neural Networks (DNNs) is *learning*, also known as *training*, which refers to the process of changing the weight parameters, or synaptic weights, of a deep neural network, for a given set of data, usually to improve a certain objective function such as difference of output from the expected ones. For software implementations of DNNs, training is often the main focus, and typically done using high-end GPUs. However, hardware implementations of DNNs such as using FPGAs (Field-Programmable Gate Arrays) [1,2] and ASIC (Application-Specific Integrated Circuit), typically lack training (i.e., backward pass), focusing on inference (i.e., forward pass) instead.

Inference-only hardware DNNs can be useful for certain applications, as it can provide lower cost and higher power/energy efficiency over software implementations of DNNs on CPU/GPUs. At the same time, it is also obivious that hardware DNNs fortified with training capability can be more useful and have more applications. For instance, on-chip training hardware DNNs can provide better customization for different users and/or environments, as well as higher fault-tolerance against device faults.

Despite the advantages, designing on-chip training hardware DNNs has some challenges such as how to design a common datapath that can be well utilized for both inference and training. An important challenge in this respect is that often training and inference have widely different precision requirements for arithmetic operations. The main computation in both inference and training of DNNs is the multiply-and-accumulate (MAC) operation, but while inference typically

requires 8-bit precision even for large datasets, training seems to require 16-bit precision [3]. This means that if one uses 16-bit precision operations for training, the same hardware DNN may not achieve best efficiency during inference.

This paper present learning-inference architecture to support both learning and inference acceleration of deep neural network. To accelerate inference, we pack two MAC operations into one DSP block like SIMD operation, which we call *Double MAC*. In case of training, we use one DSP block for one MAC operation with 16-bit precision multiplication to more accurate computation.

We also implement our architecture on a ZC706 FPGA board with Caffe framework to demonstrate our accelerator. With Caffe framework, we can show full process of deep neural network including training and inference. In particular, we focus on image classification using dataset MNIST, CIFAR-10 and ILSVRC [4–6].

In this paper we present a novel learning-inference architecture using Double MAC, featuring the following contributions.

- We consider training for FPGA implementation of deep neural network.
- We present the datapath of Double MAC for supporting both training and inference.
- We implement learning-inference architecture with Caffe framework.

The rest of this thesis is organized as follows. After reviewing the background and previous work in Chapter II. We explain our mapping deep neural network to FPGA in Chapter III. The experimental results presented in Chapter IV. Then we present our futurework in Chapter V. Finally, we conclude the paper in Chapter VI.

CHAPTER II

BACKGROUND AND PREVIOUS WORK

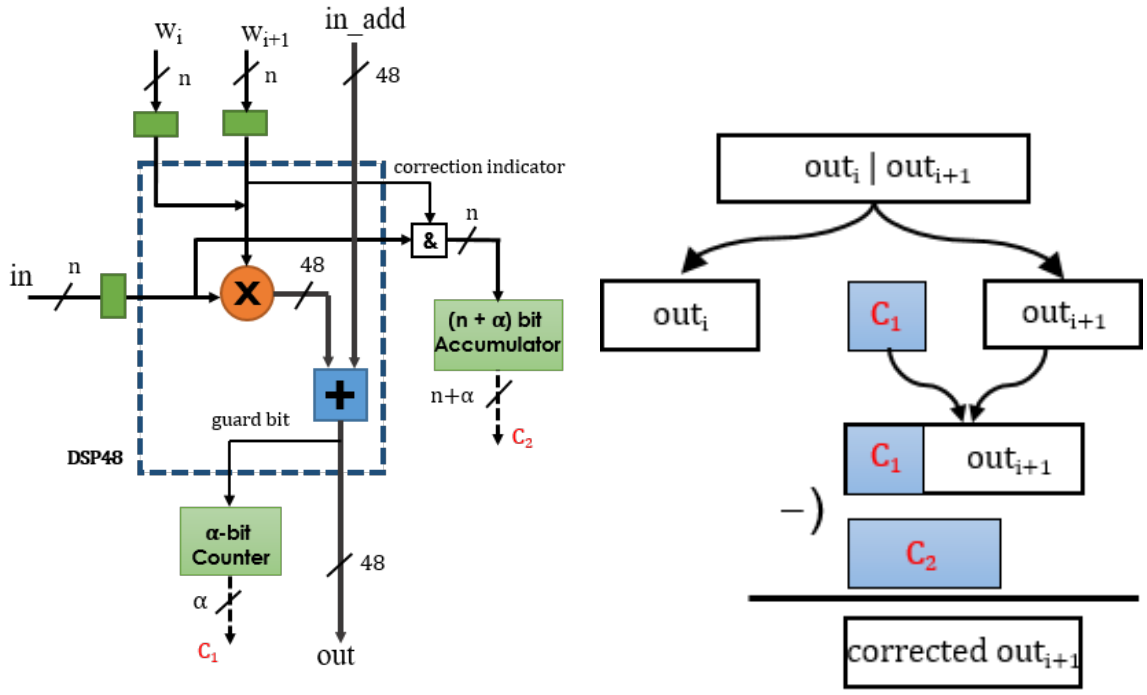
2.1 Background

2.1.1 Fixed Point Arithmetic

We implement our Double MAC in fixed point arithmetic because floating point arithmetic requires too large FPGA resources. In order to implement floating point 32-bit multiplier, 5 DSPs and additional resources are needed [1]. On the other hand, implementation of fixed-point 32-bit multiplier require 4 DSPs and implementation of fixed-point 16-bit multiplier needs 1 DSP [1, 7]. We explain this in more detail in next section.

2.1.2 DSP Block in State of The Art FPGAs

State of the art FPGA chips have special block such as BRAM (Block Random Access Memory) and DSP (Digital Signal Processor) block to improve it's performance. Between BRAM and DSP block, we focus on optimization of DSP block. Xilinx designs their FPGA chips with custom DSP48 block. They design two type of DSP block which are DSP48E1 and DSP48E2 block. In this papers, we deal with the DSP48E1 because the DSP48E1 can cover most series of Xilinx FPGAs and the DSP48E1 block can calculate up to 25-bit $\times$ 18-bit for one cycle.



(a) The datapath of our architecture (b) The method of modification

Figure 2.1: Our proposed Double MAC architecure

2.2 Previous Work

2.2.1 The Datapath of Our Architecture

We explained how to claculate unsigned-unsigned multiplication and signed-unsinged multiplication in previous work [7]. In deep neural network field, input value can be from 0 to 1 in floating point arithmetic and weights value can be from -1 to 1 in floating point arithmetic. Therefore, we consider signed-unsigned multiplication for implementation of FPGA.

In this section, we present our architecture Double MAC (See Figure 2.1a). Double MAC has n-bit input, W_i and W_{i+1}, that also has 48-bit input for addition and output. We implement one multiplication and one addition in DSP48E1 block. Sign-bit of W_{i+1} use as correction indicator because signed-unsigned case only needs correction steps. If W_{i+1} is negative, input value accumulates $(n + \alpha)$bit accumulator. $(n + \alpha)$bit accumulator output C_2 $(n + \alpha)$bit have. α-bit counter accumulates guard bit every cycle, that output C_1 α-bit have.

2.2.2 The Method of Modification

Figure 2.1b illustrate how to modify incorrect result of out_{i+1}. If correction indicator bit is 1, out_i is correct but out_{i+1} is incorrect. Thus, modification is needed so we combine C_1 with

out_{i+1} then it subtract C_2 after shifting C_2 to most of left. After this process, result of out_{i+1} is correct.

The DSP48E1 block can perform one multiplication and one addition for one cycle, that originally design two's-complement multiplier. However, we can use the DSP48E1 as unsigned-unsigned or signed-unsigned multiplier by configuring the DSP48E1 block. For instance, in order to pack two MAC operations into one DSP block, $(3n + 1) \times n$-bit are needed. In case of DSP48E1, The biggest n we can choose is 8-bit because of $(3 \times 8 + 1) = 25$.

CHAPTER III

MAPPING DEEP NEURAL NETWORK TO FPGA

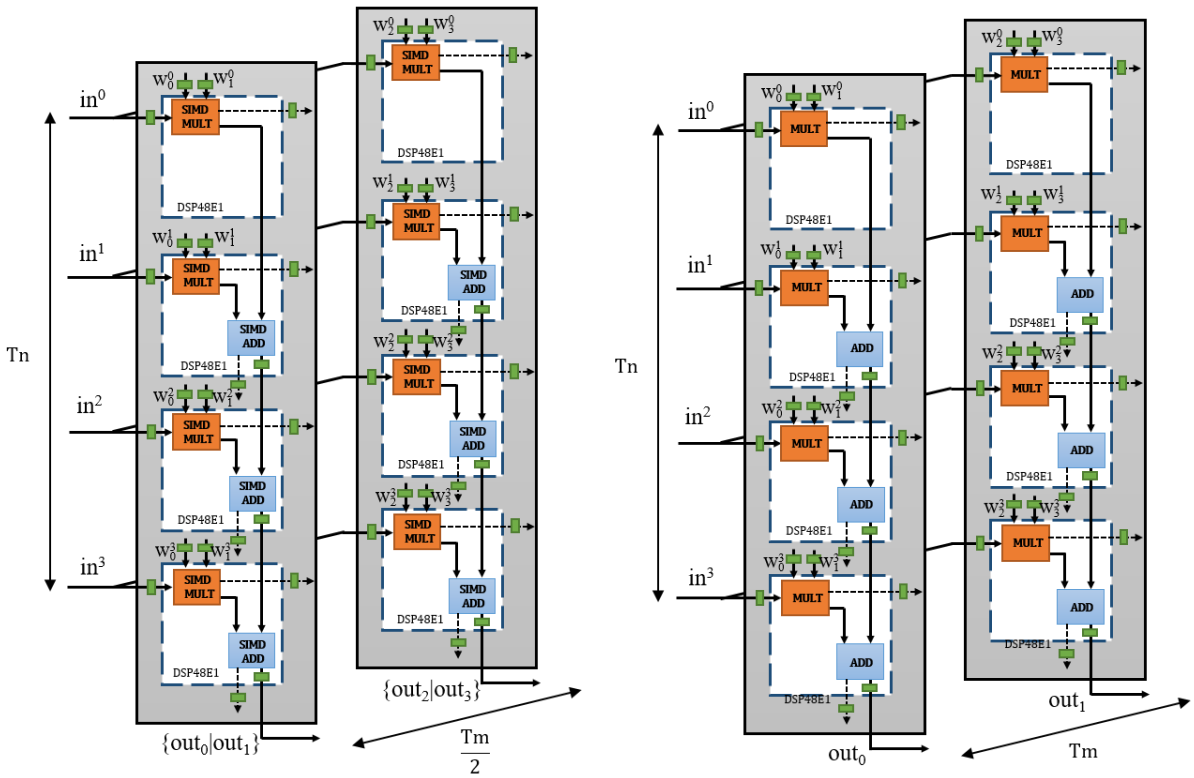
3.1 Our Architecture

3.1.1 Implementation of MAC Array Using Double MAC

Figure 3.1a illustrates our SIMD (Single Instruction Multiple Data) MAC array using Double MAC. We implement MAC array with pipelined structure. Each of inputs (in^0, in^1, in^2, in^3) is shared with each of column MAC structure (In Figure 3.1a, we have two column MAC structure). Each of weights (W_n^n) is fed separately to each of Double MAC unit. The output data from SIMD MULT (First MAC unit) or SIMD ADD (Others) is going down, the data is accumulated. Then, we can get output from the lowest Double MACs.

3.1.2 The Datapath of Double MAC for Supporting Both Training and Inference

Figure 3.1 illustrates our SIMD unit in Double MAC. Figure 3.1a shows the datapath of Double MAC for 8-bit double mode and Figure 3.1b present Double MAC for 16-bit single mode. those two share the same computation unit and the datapath. To share the datapath for



(a) 8-bit Double MAC (b) 16-bit Single MAC

Figure 3.1: Our proposed Double MAC array for supporting 8-bit and 16-bit

Table 3.1: Vector concatenation

Mode	16-bit Single MAC	8-bit Double MAC
Select	0	1
Input	input[15:0]	input[7:0] (input[15:8] don't care)
Weights	$\{W_i \| W_{i+1}\}$	$\{W_i \| 9'b0 \| W_{i+1}\}$ (9-bit including guard bit)
Output	out_1, out_2 (each output is 16-bit)	$\{out_0 \| out_1\}$, $\{out_2 \| out_3\}$ (each output is 8-bit)

8-bit and 16-bit MAC operations, we extend input wire from 8-bit to 16-bit and each input W_i and W_{i+1} can combined together and it use for one 16-bit precision weight.

In order to support two mode, vector concatenation is needed (See Table 3.1). In case of 16-bit Single MAC mode, select signal have to set 0. We can use 16-bit input and weight combined with W_i and W_{i+1}. Then, we can get the output out_n and each of outputs is 16-bit.

On the other hand, In case of 8-bit Double MAC mode, select signal have to set 1. In this case, we only use input from 0 to 7 bit and the input from 8 to 15 is ignored. We also use weights combined with W_i, 9-bit 0 and $W_i + 1$. Then, we can get the output combined with out_0 and out_1, out_2 and out_3 respectively. $out_n|out_{n+1}$ is 16-bit and each of out_n is 8-bit so each of the outputs has two outputs.

3.1.3 RTL-HLS Hybrid Architecture

Figure 3.2 illustrates our buffer structure. In order to control Double MAC, we design RTL-HLS (High-Level Synthesis) hybrid architecture. Because of speed difference between Double MAC (RTL) and ARM processor, data buffers are needed.

We design RTL-HLS hybrid architecture. Our Double MAC can perform T_n vector and $T_m \times n$ matrix multiplication and by repeating n cycles, that can perform $T_n \times n$ matrix and $T_m \times n$ matrix multiplication.

We design Double MAC with pipelined structure so we need to control inputs and weights every cycle. In order control inputs and weights every cycle, we make input register, multiplexer and counter. If we make Double MAC can perform n cycle MAC, we need $Tm \times n$ register for input. The counter increments by one every cycle and the multiplexer choose different inputs. For instance, in case if in^0, the multiplexer put in^0cy0 to Double MAC at first cycle, that put in^0cy^1 put to Double MAC at second cycle. In case of weights, also same solution are used every cycle.

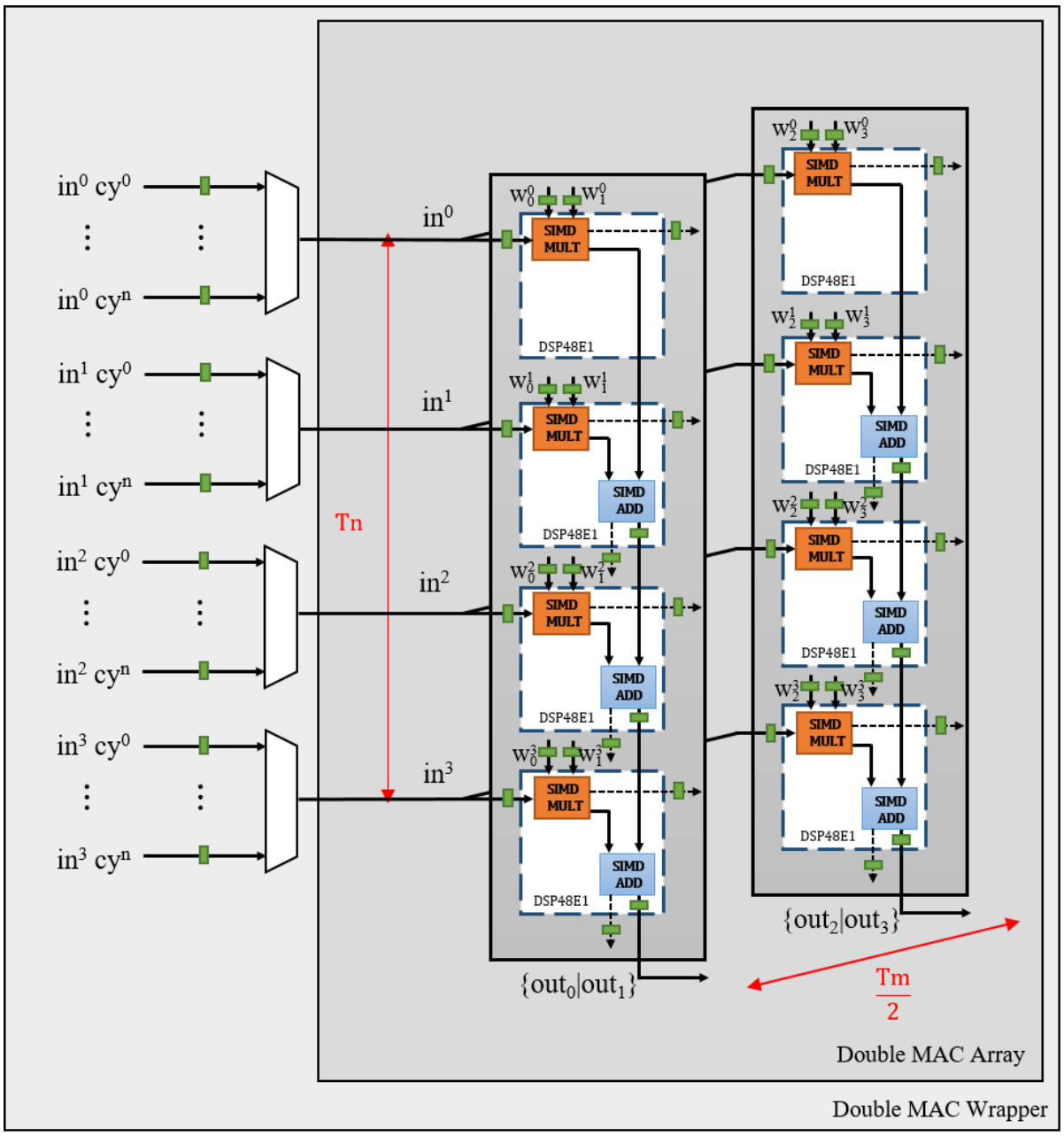


Figure 3.2: Our buffer structure for Double MAC array

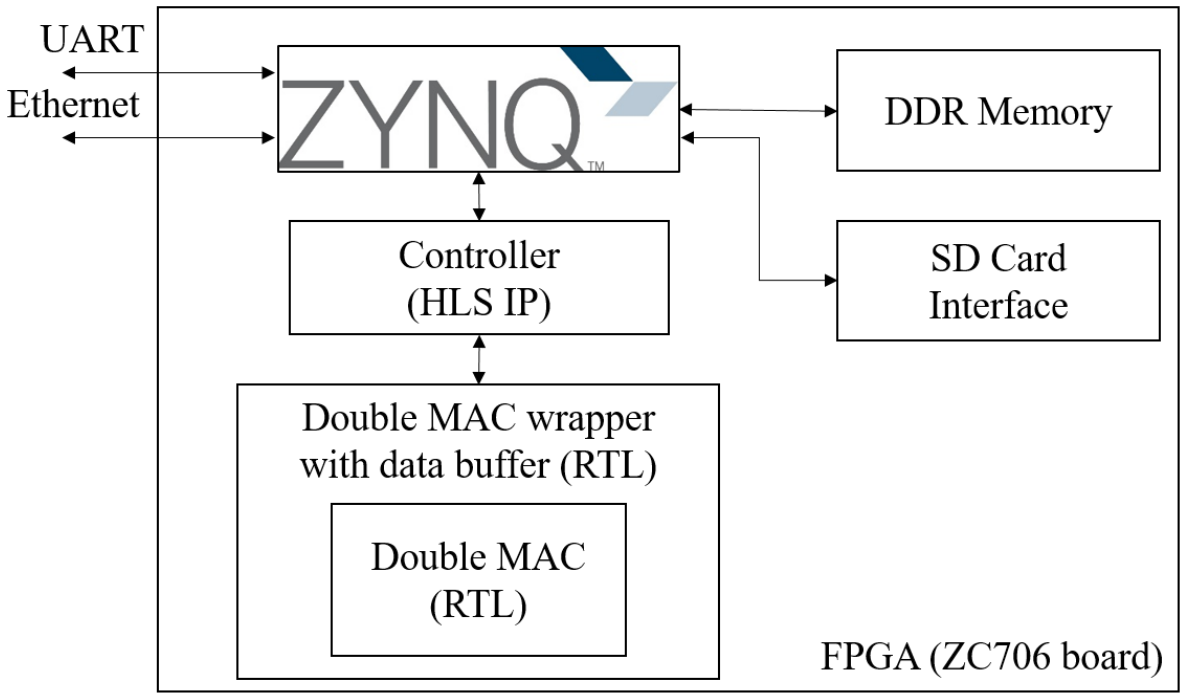


Figure 3.3: System block diagram

3.1.4 System Block Diagram

Figure 3.3 illustrates our system block diagram. Our system has our accelerator Double MAC, controller generate by HLS and ZYNQ processing system. ZYNQ processing system has ARM processor, DDR memory controller and SD card controller. We store Linux boot file and images to SD card. When booting the system, zynq processing system boot from SD card. ZYNQ processing system also has UART and Ethernet interface so we can connect the system through UART and Ethernet.

3.2 Caffe Framework on FPGA

3.2.1 Caffe Structure with FPGA

Caffe framework is one of the powerful framework for deep neural network developed by the Berkeley Vision and Learning Center (BVLS) [8]. We run Caffe framework on Linux on ZYNQ FPGA board, and that has ARM processor. Consequently, we can use Ubuntu Linux compiled by ARM compiler. Then, we compile and install Caffe framework using ARM compiler on ZYNQ FPGA board. Caffe applications are usually written by Python so we use Python interface. Figure 3.4 illustrates Caffe structure with FPGA and Python interface.

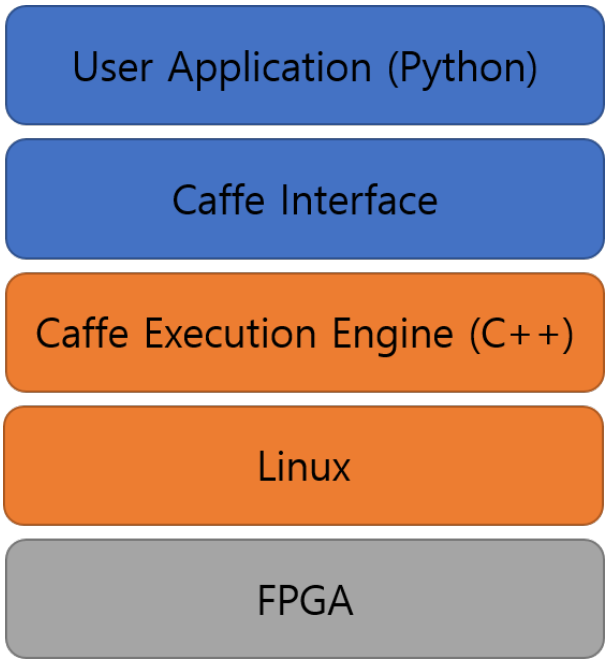


Figure 3.4: Caffe structure with FPGA

3.2.2 Converting GEMM to Common Type of Matrix Multiplication

Caffe framework performs matrix multiplication using general matrix-matrix multiplication (GEMM) based on BLAS (Basic Linear Algebra Subprograms). To make implementation of FPGA resource-efficient, we convert GEMM to common type of matrix multiplication. GEMM based on BLAS is too complicated to make implementation of FPGA.

$$\mathbf{y} \leftarrow \alpha x + \mathbf{y} \tag{III.1}$$

$$\mathbf{y} \leftarrow \alpha \mathbf{A} x + \beta \mathbf{y} \tag{III.2}$$

$$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C} \tag{III.3}$$

Level 1 of BLAS consists of vector operations (See (III.1)) and level 2 of BLAS contains matrix-vector multiplication (See (III.2)) [9, 10]. Level 3 of BLAS contains matrix-matrix multiplication (See (III.3)) [11]. Caffe framework can support all type of BLAS but only use level 3 of BLAS.

Caffe framework is originally software solution to implement application of deep neural network. In other words, Caffe framework not fit for implementation of FPGA so we modify

Table 3.2: Matrix multiplication parameters

Parameters	Description
N_i	Input A and output C matrix width
N_j	Input B and output C matrix matrix height
N_k	Input A and input B matrix matrix height and width
T_i, T_j, T_k	Tile parameter
μ_i, μ_j	Parameters to be fixed to HW for T_i and T_j
μ_k	Parameter to be fixed to HW for T_k, it takes T_k cycles
T_n	Input parameter for Double MAC
T_m	Output parameter for Double MAC
n	# of cycles

GEMM operations to that only supports level 3 of BLAS. For most of deep neural network, level 3 of BLAS is sufficient to be implemented. In case of applications that handle vectors, level 3 of BLAS is enough to be implemented because most of applications using Caffe framework use the batch process. Thus, even though applications that handle vectors part of computations will become matrix multiplication.

3.2.3 Weights Update

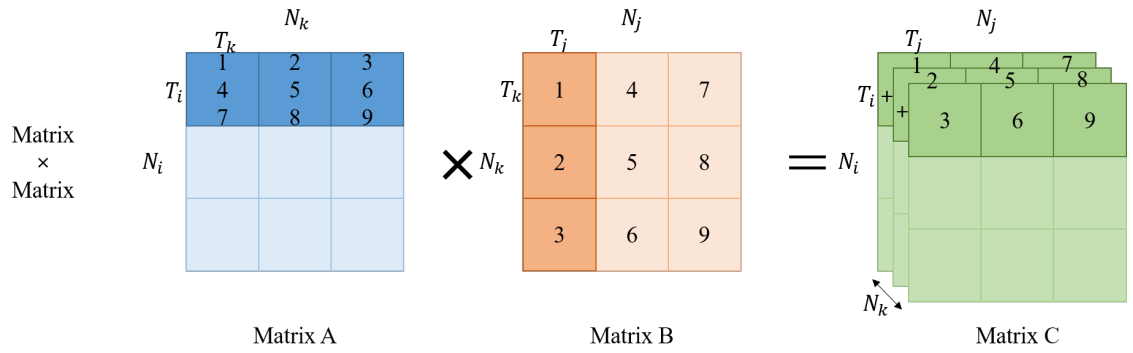
$$\mathbf{C} \leftarrow \mathbf{AB} + \beta\mathbf{C}(\forall\beta \in [0, 1]) \tag{III.4}$$

In order to reduce resource of FPGA, we redesign level 3 of BLAS that beta (β) can be only 0 or 1 (See (III.4)). Which means it can write code using if statement, not multiplication operations. We can implement part of weights update using beta (β) using if statement. We also eliminate alpha (α) operations to reduce multiplication operations.

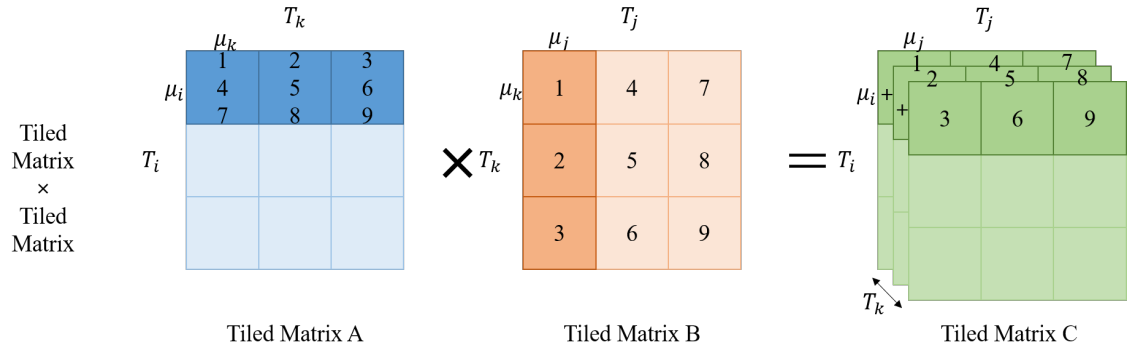
3.2.4 Tiled Matrix Multiplication

Table 3.2 shows matrix multiplication parameters. N_i, N_j and N_k is whole matrix multiplication parameters defined by applications. T_i, T_j and T_k is tile parameters to be computed with one HW call. μ_i, μ_j and μ_k parameters need to fix to generate Double MAC. μ_i and μ_k is matched to $T_m \times n$ weight matrix and μ_k and μ_j is matched to $T_n \times n$ input matrix.

Figure 3.5 illustrates two-level tiled matrix multiplication scheme with parameters (T and μ). We design tiled matrix multiplication with tile parameters (T and μ). Whole matrix multiplication is running in ARM processor in FPGA. Tiled part of matrix multiplication with tile parameters (T_i, T_j and T_k) is running in controller generated from HLS. Double MAC can perform tiled part of matrix multiplication with mu parameters (μ_i, μ_j and μ_k).



(a) Matrix multiplication with T parameters



(b) Tiled matrix multiplication with μ parameters

Figure 3.5: Two-level tiled matrix multiplication scheme with parameters (T and μ)

```
for (int ni = 0; ni < Ni; ni += Ti)                                   | SW running in ARM
  for (int nj = 0; nj < Nj; nj += Tj)
    for (int nk = 0; nk < Nk; nk += Tk)
      read block TA, TB, TC(ti, tj);
      for (int ti = 0; ti < min(ni+Ti, Ni); ti += Ui)                 | Controller (HLS)
        for (int tj = 0; tj < min(tj+Tj, Nj); tj += Uj)
          for (int tk = 0; tk < min(tk+Tk, Nk); tk += Uk)
            read block UA, UB, UC(ui, uj);
            for (int i = 0; i < min(ti+Ui, Ti); i++)                  | Data computation using Double MAC
              for (int j = 0; j < min(tj+Uj, Tj) j++)
                Dtype acc = 0.0;
                for (int k = 0; k < min(tk+Uk, Tk); k++)
                  acc += UA[i*Uk +k]*UB[k*Uj + j];
                UC[i*Uj + j] = UC_in[i*Uj + j] + acc;
            write block UC(ui, uj);
      write block TC(ti, tj);
```

Figure 3.6: Matrix multiplication code in C with tile parameters (T and μ).

Because of limitations of FPGA resource whole part of matrix multiplication can not be implemented in FPGA. Thus, we implement tiled matrix multiplication in FPGA which can be synthesized. Our Double MAC can compute part of data computation (See Figure 3.6).

CHAPTER IV

EXPERIMENTS

4.1 Experiment Setup

We have implemented our Double MAC in Verilog and our target device is XC7Z045FFG900-2 on Xilinx Zynq-7000 SoC ZC706 Evaluation Kit (See Figure 4.1). We use Xilinx Vivado 2017.4 as integrator and Xilinx Vivado HLS 2017.4 for generating controller. In order to generate Linux kernel, Xilinx PetaLinux 2017.4 is used. Then we run Ubuntu 16.04 on ZC706 SoC board as embedded operating system. We store Linux kernel generated by Petalinux, Ubuntu root file system, Caffe framework and dataset to SD card.

4.2 Synthesis Result

Table 4.1 shows utilization of resource. We limit DSP usage up to 80% because other component may use DSP blocks. Parallelization factor of our accelerator (T_n, T_m) is (24, 60) so we can calculate the number of DSPs as $T_n \times \frac{T_m}{2}$. Our target device has 900 DSPs available and we utilize 720 DSPs. In order to extend 8-bit fixed Double MAC to 8-b/16-b fixed Double/Single MAC, FFs and LUTs need more 50% resources respectively.

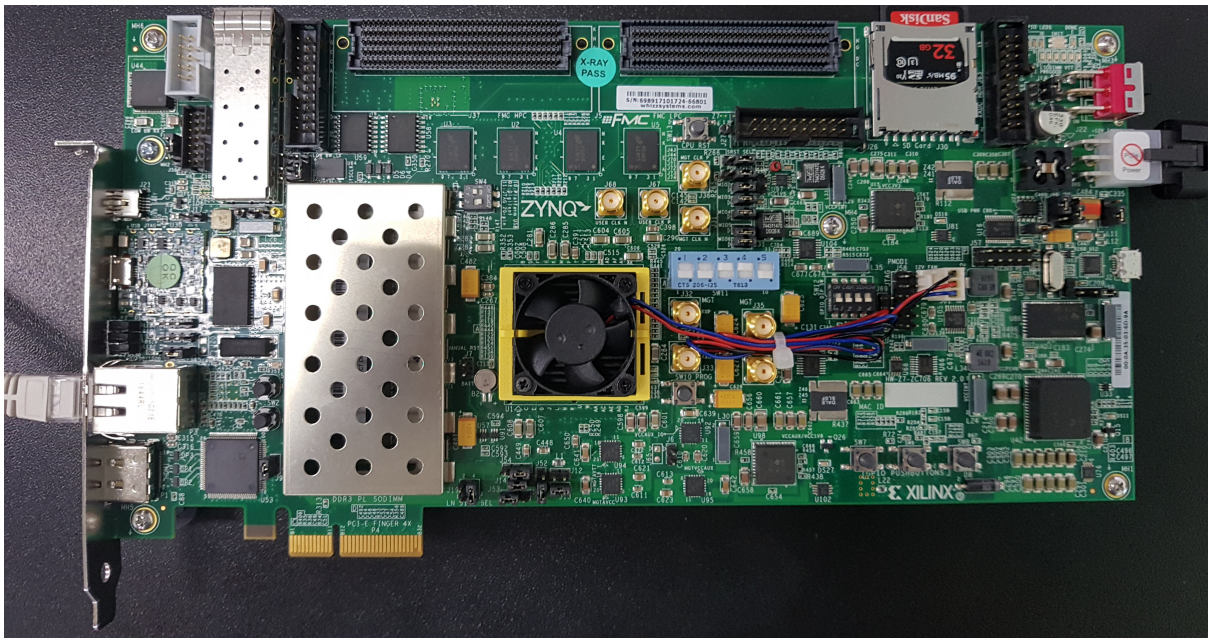


Figure 4.1: Xilinx Zynq-7000 SoC ZC706 Evaluation Kit

Table 4.1: Resource utilization

	(T_n, T_m)	DSP48E	FF	LUT
8-b fixed Double MAC	(24, 60)	80.00%	4.66%	10.34%
8-b/16-b fixed Double/Single MAC	(24, 60)	80.00%	7.17%	15.91%

4.3 Validation of Performance Model

$$\#\text{ of tile per layer (16-b Single MAC)} = \left\lceil \frac{N_i}{\mu_i} \right\rceil \times \left\lceil \frac{N_j}{\mu_j} \right\rceil \times \left\lceil \frac{N_k}{\mu_k} \right\rceil \tag{IV.1}$$

$$\#\text{ of computation cycles per tile (16-b Single MAC)} = 2 + 2 \times (T_n + 2) + \mu_j cycles \tag{IV.2}$$

$$\#\text{ of total cycles per layer} = \#\text{ of tile per layer} \times \#\text{ of computation cycles per tile} \tag{IV.3}$$

$$\text{computation time per layer} = \#\text{ of total cycles per layer} \times \text{clock period} \tag{IV.4}$$

To make validation of performance, we present a validation model. As following (IV.1), we

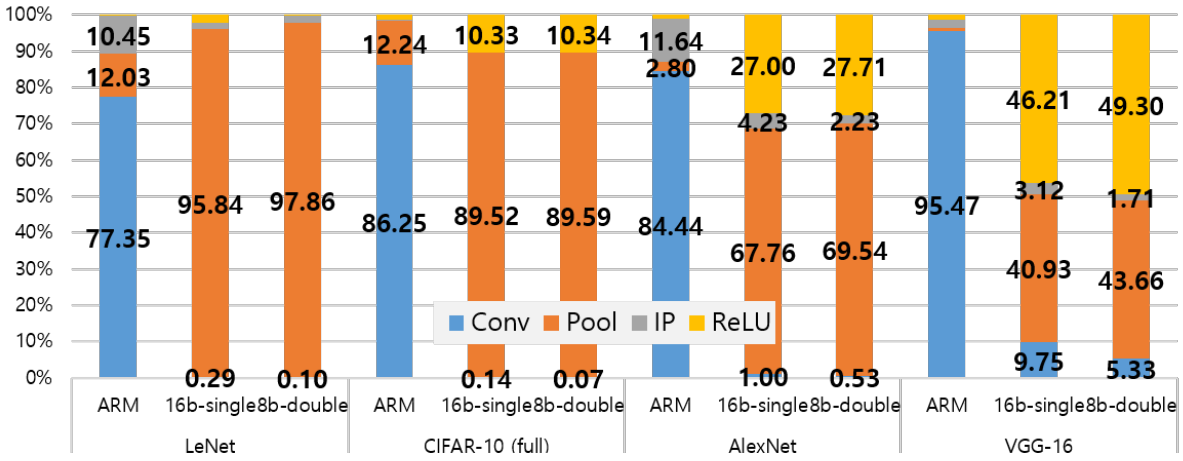


Figure 4.2: Caffe applications profiling

can calculate the number of tile per layer. The number of computation cycles can be calculated as (IV.2). Our accelerator and data transfer engine is implemented with pipelined structure so our accelerator need 2 cycles for load and store data, that has T_n pipeline depth so T_n cycles for computation are needed. In order to control our accelerator, reset and done cycle is also needed so it takes $2 \times T_n + 2$ cycles. The pipeline structure has prologue and epilogue so it takes 2 times. Additionally, μ_j cycles are needed because we did not parallelize matrix vector multiplication. As the number of tile per layer times computation cycles per tile, we can get the computation cycles (See (IV.3)). In order to get computation time per layer, clock period times is needed.

$$\#\text{ of tile per layer (8-b Double MAC)} = \left\lceil \frac{N_i}{2 \times \mu_i} \right\rceil \times \left\lceil \frac{N_j}{\mu_j} \right\rceil \times \left\lceil \frac{N_k}{\mu_k} \right\rceil \tag{IV.5}$$

$$\#\text{ of computation cycles per tile (8-b Double MAC)} = 2 + 2 \times (T_n + 3) + \mu_j cycles \tag{IV.6}$$

In case of Double MAC, we have $2 \times \mu_i$ factor, because mapping μ_i factor to T_m factor. Thus, we can calculate the number of tile per layer as (IV.5). In (IV.6), we need one more cycle for modification.

4.4 Performance

We profile each application (MNIST, CIFAR-10 and ImageNet) and network (LeNet, CIFAR-10 (full), AlexNet and VGG-16) [5,12–14]. Comparing 8-b Double MAC with 16-b Single MAC, 8-bit Double MAC doubled performance. Figure 4.2 illustrates each function of run time ratio.

Before accelerating convolution layers, most of time period occupies convolution layers. After accelerating convolution layers, pooling layer occupies most of time period. In VGG-16, activation function ReLU also takes more then 40% time period.

CHAPTER V

FUTURE WORK

First of all, in this work, we focus on mapping matrix multiplication to Double MAC. One of the limitation of this work, Each tiling parameters is not optimal. In order to enhance performance, we will find optimal factor (T_n, T_m) for each application.

Secondly, we implemented matrix multiplication using matrix vector multiplication and mapped the matrix vector multiplication to our Double MAC. In this approach, matrix multiplication takes N_j cycles. To over come this weakness, we will parallelize each of matrix vector multiplication.

Lastly, Caffe framework converts image to column to perform convolution operations using matrix multiplication [15]. One of the way to eliminate this overhead, we can implement convolution operations using our Double MAC instead of matrix multiplication.

CHAPTER VI

CONCLUSION

In this paper, we present novel learning-inference architecture using Double MAC and datapath for supporting both training and inference. Our datapath is highly efficient way to perform training and inference without FPGA reconfiguration. Though our datapath need more 50% FFs and LUTs, our accelerator can support training and inference. We also present how to map matrix multiplication to our Double MAC.

We convert GEMM based BLAS to common type of matrix multiplication and optimize GEMM operations to fit GEMM operations to FPGA. Originally, Caffe framework is software solution to implement emerging application such as deep neural network. Because the GEMM is not fit to FPGA, we optimize the GEMM operations that eliminate alpha (α) operations and modify beta (β) operations can be only 0 or 1.

We achieved end-to-end implementation by running Caffe framework on ZC706 SoC board. We have ran our accelerator with Caffe framework after replacing GEMM with it. We profiled performance of applications (MNIST, CIFAR-10 and ImageNet) including convolution layers, pool layers and activation layers.

References

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), pp. 161–170, ACM, 2015. 1, 3

[2] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), pp. 26–35, ACM, 2016. 1

[3] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," *CoRR*, vol. abs/1711.02213, 2017. 2

[4] J. M. Alonso-Weber, M. P. Sesmero, G. Gutierrez, A. Ledezma, and A. Sanchis, "Handwritten digit recognition with pattern transformations and neural network averaging," in *Proceedings of the 23rd International Conference on Artificial Neural Networks and Machine Learning — ICANN 2013 - Volume 8131*, (Berlin, Heidelberg), pp. 335–342, Springer-Verlag, 2013. 2

[5] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009. 2, 17

[6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. 2

[7] D. Nguyen, D. Kim, and J. Lee, "Double mac: Doubling the performance of convolutional neural networks on modern fpgas," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, (3001 Leuven, Belgium, Belgium), pp. 890–893, European Design and Automation Association, 2017. 3, 4

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014. 10

[9] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, pp. 308–323, Sept. 1979. 11

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, pp. 1–17, Mar. 1988. 11

[11] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, pp. 1–17, Mar. 1990. 11

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012. 17

[13] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. 17

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, pp. 2278–2324, 1998. 17

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. 19